3DO M2 Link/Dump Programmer's Guide

Version 2.7 – June 1996

Contents

Preface

About This Document

This document provides:

- A tutorial-style introduction to using dynamic linked libraries
- References pages for Link3do and dump3do

About the 3DO M2 Link/Dump Programmer's Guide

The 3DO M2 Link/Dump Programmer's Guide describes the linker (Link3do), an MPW tool used to link objects and libraries together in order to create 3DO applications and dynamic link libraries (Dlls). Dump3do, an MPW tool used in conjunction with Link3do, generates a text file of the symbols exported by the DLL file.

Audience

This document is for C programmers preparing titles for the M2 Development System.

How this Document is Organized

This manual is organized as follows:

- Chapter 1, "Using Link3do and Dump3do ," provides a tutorial-style introduction to using dynamic linked libraries.
- Chapter 2, "Reference Pages ," contains references pages for Link3do and Dump3do.

Typographical Conventions

The following typographical conventions are used in this book:

Item	Example
code example	`Scene_GetStatic(scene)`
procedure name	`Char_TotalTransform()`
new term or emphasis	In M2, *characters* are objects that can be displayed on the screen.
file or folder name	The *remote* folder, the *demo.scr* file.

Chapter 1

Using Link3do and Dump3do

Introduction

The linker (Link3do) is an MPW tool used to link objects and libraries together in order to create 3DO applications and dynamic link libraries (Dlls). Dump3do is an MPW tool that generates a text file of the symbols exported by the DLL file.

About This Chapter

This chapter discusses the 3DO Link/Dump tools in general terms, then presents a walk-through tutorial that describes how to create and use Dlls.

Topics include:

Overview

Dynamic linked libraries (Dlls) make it possible for an executable to reference symbols at run-time. Dlls contain code that can be shared among tasks and loaded by the operating system on demand.

The advantages to using Dlls over static linking include:

- Smaller application size because Dll code is not statically appended to the executable
- Loading of code that might not be executed can be delayed until the program explicitly requests it. This is called demand loading.

The disadvantage to using Dlls over static linking is that load-time is increased because the loader must load the shared code/data and fix up referenced symbols at load-time/run-time.

Creating a DLL

To use a Dll, you need to create a definitions file that defines the symbols the object will import and/or export. This file lists the symbols a source file needs to reference (import) or export. The linker uses this file to resolve symbols at link time and include information in the binary for use by the loader. The syntax for the definitions file is descrtibed in Example 1-2.

The definitions file must state the Dll's library ID and list what symbols defined in the object are to be exported. A unique library ID, or "module ordinal" number, is needed to enable the loader to identify this Dll. You Link the object with the definitions file using the -x option; the resulting binaryis the Dll.

Importing From a DLL

You can specify imports in two ways:

- You can create a definitions file that explicitly specifies what symbols to import and from what Dlls.
- You can link with the DLL itself. In this case, the linker simply searches through the files specified on the command line for objects (DLLs) that contain the symbols as exports and automaticallly imports the symbols it needs from the Dlls.

The advantage of linking through an imports definitions file is that you don't have to list all your DLLs on the link command line. This is particularly convenient when you want to import something from a DLL you don't have. In that case you create a definitions file with the symbols you want to import, including the library number and symbol numbers you want to import.

The library ordinal identifies the unique library ID, while the symbol ordinal identifies the symbol within that library.

Flags (see Example 1-1) can be specified to tell the loader when and how to import the Dll.

Example 1-1 *Flags.*

```
IMPORT_NOW - load the DLL at the time the exceutable is loaded
         (import at load-time)
REIMPORT_ALLOWED - allow the DLL to be reloaded
IMPORT_ON_DEMAND - wait to load the DLL until a system call is
         issued from the executable to do so (import at run-time)
IMPORT_FLAG <number> - arbitrary flag number (for future use)
```

Example 1-2 defines the syntax grammer for definitions files. In this example,

- [x] - x is optional
- <x> - x is replaced by literal
- x* - x repeated any number of times
- x | y - either x or y

Example 1-2 *Syntax grammer for definitions files.*

```
definitions_statement*
definitions_statement->
!<comment>
export_definitions
import_definitions
import_definitions->
IMPORTS imports [imports_flags]
export_definitions->
MODULE <dll_ordinal> EXPORTS exports
exports->
<export_ordinal>=<symbol_name>
imports->
<symbol_name>=<dll_ordinal>.<export_ordinal>
imports_flag->
flag_name dll*
dll->
<dll_name> | <dll_ordinal>
flag_name->
IMPORT_FLAG <number>
IMPORT_NOW
REIMPORT_ALLOWED
IMPORT_ON_DEMAND
```

Creating and using Dlls

This section describes how to create and use Dlls. An application, t, wants to import symbols from two Dlls: t2.dll and t3.dll. The process involves three basic steps which, for convenience, have been labeled as such:

- Step 1: Create t2.dll
- Step 2: Create t3.dll
- Step 3: Create t.elf, Importing From t2.dll and t3.dll

Each of the Dlls uses a "definitions" file that defines what that Dll will export. The application t also uses a definitions file to show what it wants to import from t2.dll, and to specify how to import t3.

However, no definitions files are needed if an application is linked with a Dll and the default import flags are sufficient (the default is IMPORT_NOW—to load the Dll when the application is loaded).

Note: *When using createm2make, the linker options below would be automatically generated into the makefile when the option to create a DLL" is used.*

Assume that we've already created objects t.o, t2.o and t3.o (see compiler options for how to create objects).

Step 1: Create t2.dll

Let's look at the link line for t2.dll.

```
link3do -r -x t2.def t2.o -Hversion=10 -Hrevision=2 -o t2.dll
```

In this example:

- -r makes it relocatable
- -x says here's the definitions file for this object
- t2.o is the object, containing the symbols to be exported
- -Hversion=10 specifies to use version 10 of the application
- -Hrevision=2 specifies to use revision 2 of the application
- t2.dll is what we call the DLL

The linker takes the object t2.o, reads the definitins, and creates a DLL called t2.dll. It stores the library number, and what it exports, in the export section of that file at the ordinal numbers specified in the sample definitions file.

Example 1-3 is the definitions file for this DLL. It contains the export definitions for t2.c. Here we:

- define the library ordinal (or magic number) of this DLL to be 12
- export the symbol lproc2 at ordinal 0
- export the symbol ldata2 at ordinal 3

- export the symbol fproc2 at ordinal 2
- export the symbol fdata at ordinal 4

Example 1-3 *File t2.def: Export definitions for t2.c.*

```
MAGIC
12
EXPORTS
!syntax: <ord>=<symbol>
0=lproc2!ordinal 1 is skipped
3=ldata2!order does not matter
2=fproc2
4=fdata
```

Example 1-4 is the file t2.c used to create the DLL that exports the symbols ldata2, fdata2, lproc2, and fproc2.

Example 1-4 *File t2.c: Source for t2.dll.*

```
long ldata2=2;
float fdata2=2.2;

long lproc2(void) {
          return ldata2;
          }
float fproc2(void) {
          return fdata2;
          };
```

Use the following command to see the symbols exported:

```
dump3do -d t2.dll -o t2.dmp
```

Table 1-1 represents a dump of t2.dll after it was linked and built. Note the corespondence of "name" and "index" in this table with the symbols and ordinals respectively in Example 1-3.

Table 1-1 File t2.dmp: Dump of dynamic data for t2.dll.

3DO Exports (libID 12; 5 entries)				
index	**secum**	**offset**	**symidx**	**name**
0	2	x0	xa	lproc2
1	0	x0	x0	
2	2	xc	xc	fproc2
3	5	x0	xb	ldata2
4	5	x4	xd	fdata2

Step 2: Create t3.dll

```
link3do -r -x t3.def t3.o -Hversion=10 -Hrevision=3 -o t3.dll
```

Example 1-5 contains the export definitions for t3.c.

Example 1-5 *File t3.def: Expot definitions for t3.c.*

```
!t3.def
MAGIC
          13

EXPORTS
          0=vproc3
```

Example 1-6 is the file used to create the DLL t3.dll that exports the symbol vproc3.

Example 1-6 *File t3.c: Source for t3.dll.*

```
/* t3.c */

long ldata3 = 3;

void vproc3(long l) {
          ldata3 = 1;
          }
```

Use the following command to get a dump of the symbols exported:

```
dump3do -d t3.dll -o t3.dmp
```

Table 1-2 represents a dump of the Elf file t3.dll. Again note the agreement of symbol and ordinal with that in Example 1-5.

Table 1-2 *File t t3.dump: Dump of dynamic data for t3.dll.*

3DO Exports (libID 13; 1 entries)				
index	**secnum**	**offset**	**symidx**	**name**
0	2	x0	xa	vproc3

Step 3: Create t.elf, Importing From t2.dll and t3.dll

```
link3do -r -x t.def t3.dll t.o -o t.elf
```

Example 1-7 *File t3.def: export definitions for t3.c.*

```
!This is a comment

!t.def
!         This file contains import definitions for t.c

IMPORTS
          !syntax: <name>=<lib>.<ord>
          lproc2=12.0 !ordinal 1 is skipped
          ldata2=12.3 !order does not matter
          fproc2=12.2
          !exports from t3 are imported implicitly

IMPORT_NOW
          12     !set flags for how to import the Dlls
          t3.dll!can either use name or modul;e number

REIMPORT_ALLOWED
          12
```

Example 1-8 *File t.c: Source for t.elf.*

```
/* t.c

          This file imports symbols ldata2 , lproc2, and fproc2
          from the DLL t2.dll, and vproc3 from t3.dll.
*/

extern long ldata2;
extern long lproc2(void);
extern float fproc2(void);
extern void vproc3(long);

void main(void) {
          long l;
          float f;
          l = lproc2();
          if (l==ldata2)
             f = fproc2();
          vproc3(l);
          }
```

Us the fillowing command to see the symbols that were imported:

```
dump3do -r -t -d t.elf -o t.dmp
```

Table 1-3 *Dump of Elf file t.elf:*

Relocation entries for section ".text" (index2)			
offset	**relInfo**	**symbol**	**addend**
0x10	202:impre124	x00c0000:(lib 12, sym 0)	0x0
0x1a	198:impaddr16ha	x000c0003:(lib 12, sym 3)	0x0
0xle	196:impaddr161o	x000c0003: (lib 12, sym 3)	0x0
0x28	202:impre124	x000c0002: (lib 12, sym 2)	0x0
0x30	202:impre124	x000d0000: (lib 13, sym 0)	0x0

Symbol table ".symtab".

index	value	size	bind	type	section	name
0	0x0	0	loc	null	und	
1	0x0	0	loc	sect	abs	t.elf
2	0x0	0	loc	sect	.text	.text
3	0x0	0	loc	sect	.text	.text
4	0x0	0	loc	sect	.debug	.debug
5	0x0	0	loc	sect	.line	.line
6	0x0	0	loc	sect	.strtab	strtab
7	0x0	0	loc	sect	.symtab	.symtab
8	0x0	72	glob	func	.text	main
9	0x0	0	glob	null	und	lproc2
10	0x0	0	glob	null	und	ldata2
11	0x0	0	glob	null	und	fproc2
12	0x0	0	glob	null	und	vproc3

3DO Imports (2 entries)

index	name (index)	lib_code	lib_ver	lib_rev	flags
0	---(x0)	xc	x0	x0	x3
1	t3.dll (x1)	xd	xa	x3	x1

Chapter 2

Reference Pages

Introduction

This chapter contains references pages for the Link3do and Dump3do commands. These commands and a short description of each, include:

Command	Description
Dump3do	Convert libraries or object files to text.
Link3do	Link object files

Link3do

Link object files.

Synopsis

```
link3do [options]...
```

Description

Link3do links object files, dynamically linked libraries (DLLs), archive libraries, and parses import/export definitions files to create elf (symbols) files and DLLs.

Arguments

@argfile	TRead arguments from "argfile."
-	This help.
-B[d=data_base,t=text_base,i=image_base]	Set [data&bss | text | image] base.
-esymbol	Use "symbol" as entry point.
-llname	Use library lib "lname".a
-m[2 | fname]	Generate map file to standard out [fname].
-ofname	Output file (default is a.out).
-r	Generate relocations in file to keep file relative.
i	Incremental link.
-s[s]	Strip unnessary stuff from file [.comment too].
-Lpath	Add library search path.
-b[secname]=base	Set section base.
-A=alignmen	Set section alignment.
-xdef_file	Use definitions file for resolving imports/ exports.
-v	Verbose.
-g	Generate debug information in file.
-G	Generate debug information to external .sym file.
-k	Keep everything in file.
-n	Generate standard Elf file.
-D	Allow duplicate symbols.
-U	Allow undefined symbols.
-Hfield-val	Set field in the 3do header to "val."
	Field names:
	-Hname=<n>: Set the application name.
	-Hpri=<n>:Set the application priority.

-Hversion=<n>: Set the application version.

-Hrevision=<n>: Set the application revision.

-Htype=<n>: Set the application type.

-Hsubsys=<n>: Set the application subsystype.

-Htime <'MM/DD/YY HH:MM:SS' or "now>: Set the application time.

-Hosversion=<n>: Set the application osversion.

-Hosrevision=<n>: Set the application osrevision.

-Hfreespace=<n>: Set the application freespace.

-Hmaxusecs=<n>: Set the application maxusecs.

-Hflags=<n>: Set the application flags.

-Mmagicno — Use "magicno" as the magic number.

-V — Print the version of the linker.

Options set by default are listed in Table 2-1.

Table 2-1 *Default options.*

Option	Description	Value
-A	section alignment	16
-e	entry point	"_start"
-g	generate debug information if any object has a .debug section	
-b	base addresses	0
-o	output filea.out	
-s	strip the symbols from the executable unless generating debug information	

All other options are either 0 or off.

Use the following options for 3DO development:

- -r create a relocatable execuatable
- -lc link with libc.a, and any other needed libraries
- -Lpath specify the path to the libraries

Always use the -r option when creating a 3DO application. This allows the application to be relocated by the loader.

The -s option causes the linker to create small executables that load quickly. Use this option when creating executables for production.

Return Value

Zero if successful, else failure.

Implementation

Exports.

Here's a sample definitions file.

```
MAGIC
          17
  EXPORT
          !<ord>=<symbol>
          0=dproc
          2=fproc
```

Associated Files

io.h

Caveats

Elf version must match that of OS.

See Also

Dump3do

Dump3do

Convert libraries or object files to text.

Synopsis

```
dump [options] files...
```

Description

Dump elf files, objects, DLLsa, and archive libraries.

Arguments

-o<filename>	Name of output file.
-?	This Help.
-h	File headers (elf header and 3do header).
-p	Program headers.
-s	Section headers.
-t	Symbol table.
-r	Relocation entries.
-d	Dynamic sections).dynamic, .imp3do, .exp3do).
-a	Hash table contents.
-g	Debug information.
-c	Section contents.
-l	Line information.
-i	Interpret symbolic information.
-b	Binary.
-m	If library, list members.

Return Value

Zero if successful.

See Also

link3do